CHAPTER 3

第3章 分布式进程

进程的概念来自操作系统，它定义为执行中的程序。它有就绪、运行、挂起和终止等状态，具有一定的生命期。进程在操作系统中用进程块描述，是资源分配的单位。从操作系统的角度来讲，进程管理与调度是它要处理的最重要的问题。在分布式计算系统中进程还有其他重要的问题需要解决，如分布式系统中进程的组织、进程的远程执行、进程迁移、分布式对象。本章将要讨论这些问题。

3.1 分布式进程概述

虽然传统进程构成了分布式系统的基本组成单元，但实践表明，这种进程在构建分布式系统时还是显得其粒度太大。将每个进程细分为若干控制线程(Threads)的形式更为合适，用多线程进程来构建分布式系统不仅方便实现，而且能提高系统的性能。

3.1.1 进程与多线程

为了理解线程在分布式系统的作用，首先要了解什么是进程、进程与线程的关系。

1. 进程

为了执行应用程序，操作系统创建了多个虚拟处理机，每个虚拟处理机运行一个程序。这个运行中的程序称为进程，因此一个虚拟处理机就对应一个进程。为了对这些虚拟处理机进行管理和跟踪，操作系统拥有一张进程表。进程表的表目称为进程描述块，它记录着进程的 CPU 寄存器值、内存映像、打开的文件、优先权值和统计信息等。操作系统应确保独立的进程不会有意或无意地破坏其他独立进程的正确运行。操作系统需要借助硬件支持来实现这种进程间的隔离。这样就实现了多个进程透明地共享同一个 CPU 和系统的其他硬件。这种透明共享系统硬件付出的代价是进程切换的开销。

进程运行在一个执行环境中，进程的执行环境是一个资源管理单位，它包括一个地址空间、诸如信号灯那样的线程同步机制与通信接口以及高层资源（如打开的文件）。

地址空间是进程虚拟存储器的管理单位，它可以大到 2^{32} 或 2^{64} 个字节，其中幂次 32 和 64 是处理机的字长。地址空间包括多个互不邻接的区域，每个区域由它的范围（最低虚拟地址和长度）、线程的读写执行的许可和是否允许向上或向下扩展来描述。进程的地址空间至少由三个区域组成，如图 3.1 所示。

文本（Code）区域：是一个固定的不可修改的区域，存放进程的程序代码。

数据堆（Heap）区域：由存储在程序二进制文件中的数值初始化，向高虚拟地址扩展。

堆栈（Stack）区域：程序调用时用来存放返回地址等，向低虚拟地址延伸。

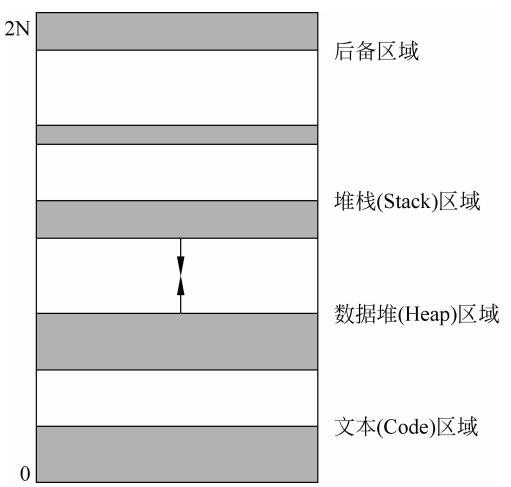


图 3.1　进程的地址空间

每次创建一个进程时，操作系统必须为它创建一个完整的独立的地址空间并对它进行初始化，即：将数据段初始化，将程序的可执行代码复制到文本段（代码段）并设置程序计数器，建立堆栈并设置堆栈指针，设置内存管理部件（Memory Management Unit，MMU）的寄存器和虚拟地址页表缓冲器（Translation Lookaside Buffer，TLB）等。在进程切换时，也存在很大的开销，除了保存 CPU 运行环境（寄存器值、程序计数器堆栈指针）外，还必须修改内存管理单元的寄存器和虚拟地址页表。如果同时运行的进程数超出设计范围，还要将某些挂起的进程转移到磁盘存储器。

在虚拟地址空间中也可以开辟进程之间和进程与内核之间共享存储器区域，共享存储器区域可用于以下目的。

（1）简化库代码的调用。库代码一般都很长，如果每个进程都单独加载，对存储器是一种浪费。替代办法是通过映射到进程地址空间的一个区域，而各个进程共享库代码的一个副本。

（2）利于数据共享和通信。为了进行任务的协调，进程之间或进程与内核之间可能需要共享数据。将共享数据映射到进程的各自区域，比两个进程通过消息传送共享数据更为有效。

（3）方便系统调用和机外处理。内核的代码和数据被映射到各个进程地址空间的同一

位置，当进程作系统调用或出现例外时，不需要新的地址映射。

2. 多线程

线程同样可以被看做程序的一部分在虚拟处理机上执行。但是线程系统一般只维护用来让多个线程共享CPU所必需的最少量信息。例如，线程环境（线程上下文）中一般只包含CPU环境（通用寄存器等）和其他一些线程管理信息，而对于多线程管理不是完全必要的信息通常被忽略。正因为如此，防止数据遭受线程错误访问的责任是由应用程序开发人员承担。

1）多线程系统的实现

线程一般是以线程包形式提供的。线程包提供创建与销毁线程操作、对同步变量（互斥变量和条件变量）操作。有三种实现线程包的基本方法，即用户模式的线程库、内核管理与调度的线程和混合形式。

（1）用户模式线程库。

用户模式线程库也称用户级线程。用户级线程的好处是：首先，创建和销毁线程操作的开销很小。由于所有线程管理工作都是在用户地址空间进行，线程创建的开销主要是建立和分配内存的开销，而销毁线程主要是释放线程堆栈占用的内存。执行这些操作的开销都不大。其次，可以通过不多的几条指令实现线程上下文的切换。基本上，只有CPU寄存器值需要保存，并将CPU的寄存器设置为切换到的线程的寄存器值。不需要改变内存的映像和刷新TLB内容与CPU统计信息。线程上下文环境的切换是发生在两个线程需要同步的时候。例如两个线程进入共享数据段，这时只能是一个在先，一个在后，进行线程切换。用户级线程的缺陷是，对于引起阻塞的系统调用会立即阻塞该线程所属的整个进程，也阻塞了所属进程中的所有线程。

（2）内核管理与调度的线程。

这种方式的开销很大，每个线程的操作（创建、销毁和同步）都必须通过系统调用由内核来执行，线程的上下文环境的切换的开销变得与进程上下文环境切换的开销同样大，用线程代替进程的优点多不复存在。

（3）混合形式。

另一种方法是用户级线程和内核级线程的混合形式，称为轻量级进程（Light Weight Process，LWP）。LWP运行在单个重量级进程的上下文环境中，每个进程可以包含多个LWP。除了LWP外，系统还提供用户级线程包，为应用程序提供创建和销毁线程等操作。重要的是线程包完全在用户空间实现，执行这些线程不需要内核干预。

线程包可由多个LWP共用。每个LWP可以运行于自己的用户级线程上。建立多线程应用程序时，首先创建线程，然后为每个线程分配一个LWP。

用户级线程与LWP一般是通过以下方式结合起来：线程包中有一个用于调度下一个线程的简单例程，在创建LWP时，LWP得到了自己的堆栈，并会得到通知去执行调度例程，该例程寻找和执行下一个线程。如果有多个LWP，每个LWP都会执行调度例程。用来跟踪和管理当前线程集的线程表是由多个LWP共享。通过在用户空间中设置互斥标志对线程表进行保护，保证对线程表进行互斥访问，即LWP间的同步不需要内核支持。

如果LWP找到了一个可运行线程，它就将上下文环境切换到该线程。如果线程由于访问互斥变量或条件变量而要被阻塞，在完成必要的管理工作之后它调用调度例程，如果找

到了一个可运行的线程，就将上下文切换到该线程。

将 LWP 与用户级线程结合起来使用的优点是：首先，线程创建、销毁和同步的开销相对较小，且不需内核干预；其次，如果每个进程有足够数量的 LWP，阻塞的系统调用不会导致这个进程被挂起；再则，应用程序不需要了解 LWP，它见到的是用户级线程；最后，通过在不同的 CPU 上执行不同的 LWP，LWP 可以在多处理机系统中方便应用。

这种混合方式的缺点如内核级线程一样，LWP 的创建和销毁的开销很大，幸好只是偶尔需要进行 LWP 创建和销毁，且完全在操作系统控制之下。

2）多线程系统的应用

在讨论线程用于分布式系统之前，考察它们在传统系统中的应用是有好处的。只有单线程的进程，一旦执行造成阻塞的系统调用，整个进程就阻塞了。多线程的好处就在这里。进程中某个线程执行导致阻塞的系统调用时，受阻塞的是该线程，而不是整个进程。进程中的其他线程仍可运行。

为此，我们来考察一下诸如电子表格这类应用程序。电子表格程序的重要特点是，要维护不同单元格之间的函数关系，而这些单元格常常位于不同的数据表格中。一旦修改某个单元格，所有有关的单元格都要自动更新。用户对某个单元格的值进行改动，会触发程序进行大量计算操作。在单线程情况下，在程序等待键盘输入时无法进行计算，在进行计算时也很难接收键盘的输入。最方便的办法是有两个线程，一个线程负责管理与用户之间的交互，另一个线程负责数据表格的更新。

在多处理机系统上执行多线程时，可以使用并行操作技术。在这种情况下，为每个线程分配一个 CPU，共享数据存放在共享存储器中，多线程可以并行执行。随着多处理机工作站的出现和价格的降低，多线程并行操作技术越来越重要并受到欢迎。这种计算机系统在客户/服务器计算范型中多用于运行服务器程序。

一个进程中只有一个线程，这种进程称为重量进程；一个进程有多个共享堆栈的线程，称为轻量进程；一个进程有多个线程，但它们有各自的堆栈，称为中量进程。轻量进程的线程缺乏保护，重量进程（线程）切换的开销最大。

3.1.2 分布式进程创建

1. 创建分布式进程

在一个新进程创建时操作系统提供了不可分割的操作。例如，UNIX 的 fork 系统调用从调用者（父进程）复制一个执行环境创建一个新进程（子进程），exec 系统调用将调用进程变换为执行指定程序的新进程。对于分布式系统，进程创建机制必须考虑多计算机的使用，因此，创建进程的支持设施分成两个独立的系统服务，即目标主机的选择和执行环境的建立。

1）目标主机选择

新进程驻留主机的选择是进程分配的决策。一般来说，进程分配策略是新进程运行在创建该进程的源主机，还是要考虑负载分担而选用其他主机。为了负载分担，可以考虑以下策略。

（1）位置策略。

确定哪个主机驻留新进程，这个决策是根据主机相对负载、机器结构及其是否拥有特殊

资源。

(2) 传输策略。

确定是本地或远程主机适合新进程,本地主机是负载轻还是重。传输策略有两种启动方法:发送者启动和接收者启动。发送者启动是创建新进程的节点负责启动传输。接收者负责启动,是某个负载轻于一个负载阈值的节点启动传输,它将接受这个新进程。

2) 执行环境建立

新进程运行主机选定后,便要为新进程建立一个执行环境,主要是地址空间。有两种方法来规定和初始化新进程的地址空间。一种方法用在地址空间是静态定义格式的场合,例如它只包含程序文本区域、数据堆区域和堆栈区域。在这种情况下,地址空间根据一张指定这些区域范围的表格来创建,地址空间由一个可执行文件进行初始化。

第二种方法是 UNIX fork 语义的一般化,父进程地址空间的各个区域被其创建的子进程所继承。一个被继承的区域或者是与父子进程共享,或者从父进程区域进行复制。当父子进程共享区域时,页面框架(内存页面)属于父进程,同时被映射到子进程的相应区域。区域共享只能用在与父子进程同在一个主机上。

在下面的进程远程执行和进程迁移两节中还会涉及主机选择和执行环境的建立。

2. 多线程的客户与服务器

线程的重要特点之一是它提供了一种方便的方法,使得在执行会导致阻塞的系统调用时不会阻塞该线程所属的整个进程。这个特点在分布式系统中很有吸引力。因为通过多个线程可以方便地将多个通信表述为同时维持多个逻辑连接。下面特别考察多线程的客户端和多线程的服务器端。

1) 多线程客户

为了确立高度分布透明性,基于广域网的分布式系统需要隐藏较长的进程间信息传播的时间。隐藏通信延时的常用方法是启动通信后立即去做其他工作。我们以大家熟悉而又常用的 Web 浏览器为例来讨论客户端多线程设计。

Web 文档是由超文本标记语言(HTML)文件组成的,HTML 文档中有纯文本文件、图像组和图标等。为了在显示器屏幕上显示 Web 文档的所有组成部分,浏览器必须与 Web 服务器建立 TCP/IP 连接,从 Web 服务器读取 Web 文档并传送给显示器显示。建立连接和从 Web 服务器读取文档数据,都可能导致阻塞。在进行远程通信时,面临的问题是每个操作可能花费很长的时间。

为了尽量隐藏通信延时,某些浏览器的工作是一面接收文档数据,一面显示部分文档。首先将文档的文本部分显示出来,以便用户查看;同时利用页面滚动之类的功能,继续获取页面的其他文件,如图像等。在收到这些文件后就立即显示,不必等待浏览器取得页面所有组件后再查看页面。用户可能感觉到 Web 浏览器同时在做多项工作。

实际上,以多线程客户的模式开发浏览器可显著地使问题得到简化。只有取得了主 HTML 文件,就可以激活多个独立线程,每个线程都与服务器建立一个独立的连接,分别读取页面的各个部分。只要在进行导致阻塞的调用时不会将这个进程挂起,与服务器建立的连接和读取页面数据的过程,就可使用标准的系统调用(可能引起阻塞)来进行服务器的程序设计。浏览器与 Web 页面服务器建立多个连接,如果服务器过载或本身性能就不高,这种多线程连接并不比单线程逐个读取页面文件的性能高多少。但是,当 Web 文档复制在多

个服务器上时，多线程浏览器就可以与不同服务器副本建立连接。通过循环策略或负载平衡技术，将浏览器对页面文件的请求转发到不同的服务器。这样便可保证整个 Web 文档完全显示所需的时间比单副本 Web 服务器要短得多。

2）多线程服务器

虽然已经看到多线程客户呈现出许多优点，但是多线程技术在分布式系统中主要还是用于服务器设计。多线程技术不仅能简化服务器代码，还使高性能服务器的开发变得更为容易。不仅在单处理机系统上如此，对目前广泛应用的多处理机系统，多线程技术就更加合适。

为了更好理解使用多线程技术编写服务器代码带来的好处，现考察文件服务器的组织结构。在进行文件读/写操作时，服务器可能会由于磁盘操作而遭受阻塞。文件服务器的一般工作方式是：等待客户对文件操作的请求输入，执行对文件的（读/写或打开/关闭）操作，将文件操作的结果返回给客户。图 3.2 所示是一个流行的文件服务器的组织结构，它有一个分发器线程和若干个工作线程。分发器线程经过对客户送来的文件操作请求进行检查后，将请求分配给某个工作线程。工作线程接收请求（如读操作）后，向文件系统发出读操作调用。执行读操作的工作线程可能被挂起，因为要花较长时间从磁盘上读文件。文件从磁盘读出后唤醒工作线程，将操作结果返回给客户。在工作线程挂起期间，如果还有对服务器的请求等待处理，便再次切换到分发器线程，由分发器线程将第二个请求分配给另一个工作线程。

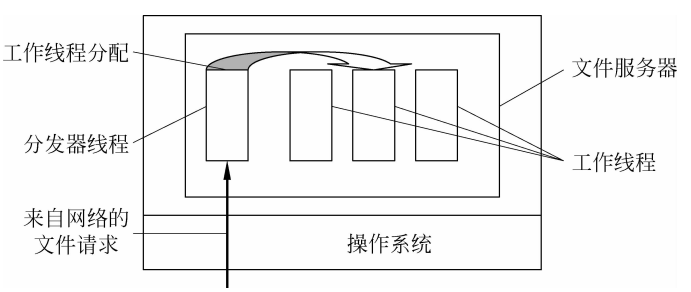


图 3.2　多线程服务器组织

如果服务器采用单线程设计，服务器线程的主循环接收、检验和执行客户的文件操作请求，直到向客户返回操作结果。而后再接收和处理客户的请求。在等待磁盘操作的时候，服务器处于空闲状态，不会对其他请求进行处理，CPU 的处理能力大大浪费了。采用多线程编写服务器代码，能提高服务器的性能，而且每个线程都是按常规方式顺序编写。总之，多线程能够保留顺序处理的思路，可使用阻塞式系统调用，能达到并行处理的目的，提高服务器的性能；单线程服务器虽然保留了使用阻塞性调用，使编程容易，但放弃了服务器的性能。

3.2　进程远程执行

在第 1 章介绍 Amoeba 时指出，在分布式计算系统中，除了产生进程外，还要为进程选择节点处理机。所谓进程远程执行，就是在集群中或网络中寻找一个或多个合适节点来执

行用户程序。在两种情况下需要远程进程执行。一种是在资源贫乏的简单用户终端，如PDA，它需要在网络上获得需要的计算资源，以完成它的计算任务；另一种情况是所谓 e-科学，如计算海洋模型，客户（科学家）需要在集群或计算网格中寻找众多的计算节点完成这样复杂而费时的计算任务。

3.2.1　远程执行概念

对进程远程执行的要求是：

（1）应有一种机构来传播空闲处理机的可用信息，或识别分布式系统中这种空闲处理机。

（2）远程执行应像进程在本地执行那样容易实现，即进程远程执行是透明的，应与位置无关。

（3）进程远程执行抢占空闲节点（工作站），当它的拥有者要求使用时，应该停止远程执行，将工作站归还其主人，实现主人优先原则。

1. 远程执行位置无关模型

在这个模型中，有两个部件：客户节点和远程服务节点。客户节点上的代理进程负责远程服务节点上远程进程执行的初始化；远程服务节点执行客户机赋予的进程。这种模型称为逻辑机模型，如图 3.3 所示，它跨越用户节点和两个远程服务节点。在一个逻辑机边界内保持文件系统、进程的父子关系和进程组的进程视图的一致，即：

（1）远程进程必须能访问驻留在源计算机上的文件系统。

（2）远程进程能接收逻辑机内任何进程发来的信号，也能将信号提供给逻辑机内任何进程。

（3）进程组保持在逻辑机内。

（4）基于树形的进程父子关系在逻辑机内必须得以保持。

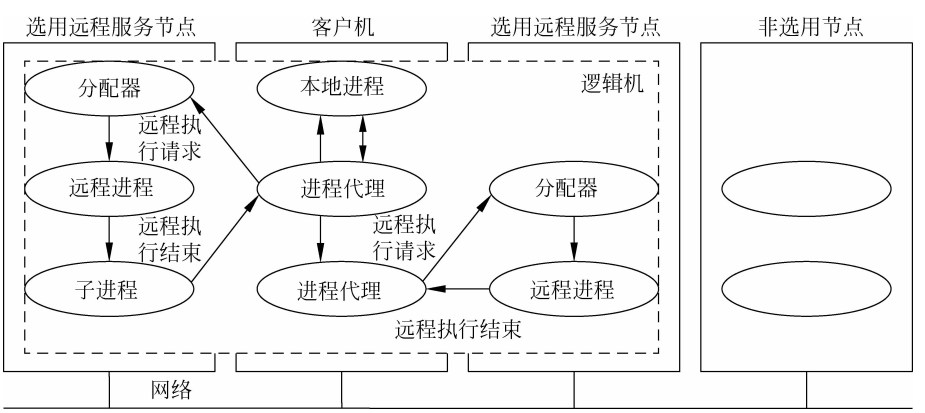


图 3.3　远程进程执行的逻辑机模型

2. 远程服务节点的选择

1）远程服务节点的加入与退出

当一个工作站空闲时或轻负载时，或许它愿意加入到远程服务节点池中去，它向资源管

理器或客户机发出一个加入消息，并提供它的信息，如处理机型号、存储器容量和操作系统类型等。资源管理器或客户机将这些信息保存在它的数据库中。当节点的主人要使用该节点或一个节点要退出远程服务节点池时，它向资源管理器或所有客户机发出一个退出消息，资源管理器或客户机从数据库中将该节点的记录删除。资源管理器是一种集中方式，驻留在一个专门的节点上，可能成为系统的性能瓶颈；在客户机建立远程服务节点数据库是一种分散方式，但占用了客户机的资源。

2）远程服务节点的选择

按照对远程服务节点的要求，客户机从资源管理器或它本身的远程服务节点数据库中查找所需要的远程服务节点，并将节点标记为已选。当进程远程执行结束后，相关远程服务节点应恢复为自由。

3. 远程执行的实现

当一个节点宣布自己为远程服务节点时，便产生一个进程执行分配器进程。客户节点发出一个连接消息到所选的远程服务节点，远程服务节点辨认后与客户节点建立一个专用的通信通道，这个通道应该是安全的（如通过 SSL 加密）。

客户节点与远程服务节点的安全通道建立后，就可以实现进程远程执行。客户节点向远程服务节点发送远程执行请求消息，请求消息包括要求执行的命令、进程执行环境（上下文）、进程组标识、信号配置、进程优先权和账户数据等。进程远程执行结束后，结果返回给客户进程。

3.2.2 远程执行 REXEC

REXEC 是一个分散的安全的集群远程执行环境，是由美国加州大学伯克利分校在 NOW 集群和 Mullennium Cluster 工作经验基础上建立起来的。

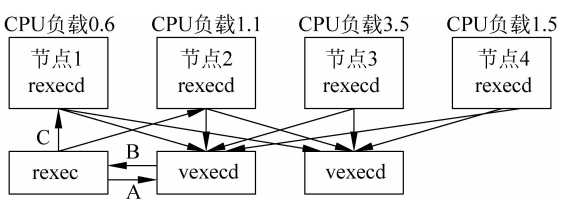
1. REXEC 结构

REXEC 结构是围绕三个实体组织的，如图 3.4 所示。这三个实体是：

（1）rexecd：运行在集群各节点上的守护进程。

（2）rexec：客户进程，用户用来在 REXEC 上执行作业。它有两个功能：根据用户要求（如轻 CPU 负载）选择节点；通过 SSL 加密，与远程服务节点的 rexecd 建立 TCP 连接，实现在远程服务节点上执行用户进程。

（3）vexecd：是一个多副本的守护进程，提供远程服务节点发现和选择功能。它不断地获得各远程节点的状态信息，其作用类似于资源管理器。



A.客户请求两个CPU负载最轻的节点；
B.节点1和节点2被选；C.客户进程在节点1和2上运行

图 3.4　REXEC 的组织结构

2. 远程服务节点的选择

(1) 客户请求远程服务节点：客户进程 rexec 向守护进程 vexecd 发出请求远程服务节点的请求，如图 3.4 中 A。例如，它需要两个轻 CPU 负载的远程服务节点。

(2) 选择远程服务节点：vexecd 从自己的数据库中，搜索到两个轻 CPU 负载的节点是节点 1 和节点 2，并将搜索结果返回给 rexec，如图 3.4 中 B。

3. 过程执行的实现

在被选远程服务节点上建立环境运行客户进程，如图 3.4 中 C。图 3.5 表示在远程服务节点上需要建立应用进程运行环境的过程，即进程的上下文和标准的 UNIX 的输入、输出和出错文件：stdin、stdout 和 stderr。其实现过程如下。

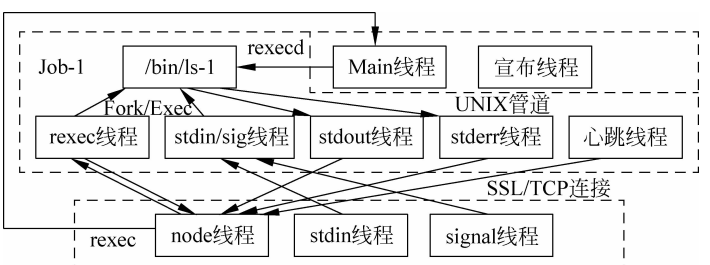


图 3.5　远程服务节点上建立环境和运行客户进程

(1) 用户本地运行环境的传播和重建。用户本地运行环境由 rexec 中的 node 线程打包发送，由 rexecd 中的 rexec 线程在用户作业派生(Forking)之后和执行(Execing)之前接收，并在远程服务节点上重建。

(2) 本地信号(signal)和 stdin 转发。本地信号(signal)和 stdin 是由 rexec 的 signal 线程和 stdin 线程转发到远程服务节点 rexecd 的 stdin/sig 线程，然后使用 signal 和 UNIX 管道将它们分发给该服务节点上的远程用户应用程序。

(3) 远程 stdout 和 stderr 转发。由 rexecd 的 stdout 线程和 stderr 线程通过 UNIX 管道将远程服务节点的 stdout 和 stderr 转发到 rexec 的 node 线程。

(4) 由本地作业控制实现对远程作业进程控制。通过转发信号(可能要进行翻译)由本地控制实现对远程作业进程的控制，即达到进程远程执行的透明性。

3.3　进程迁移

3.3.1　进程迁移概念与过程

1. 基本概念

进程迁移是早已开始研究的课题。动态进程迁移是将一个正在运行的进程挂起，它的状态从源处理机节点转移到目标处理机节点，并在目标处理机上恢复该进程运行。相对于静态放置(如进程远程执行)，进程迁移具有灵活且应用广泛的优点，例如支持动态负载平衡、系统容错、高效使用本地资源等诸多系统功能。但缺点是运行开销相对较大。目前，已经存在一些在不同程度上实现了进程迁移的系统，典型例子有 MOSIX、Chorus、Mach、

Amoeba 和 DEMOS/MP。但到目前为止，通用操作系统仍然没有普遍支持进程迁移技术。一个主要的原因是因为这些操作系统最初都是为单机运行而设计的，其进程组成及其相关数据散落在操作系统内核整个空间内，某些系统的功能设计对迁移进程的适应性较差，在这样的系统中加入透明进程迁移模块从体系结构设计上就非常复杂；另一个原因就是在机群和分布式系统广泛应用之前，没有足够的应用需求迫使操作系统厂商支持进程迁移。但是随着基于网络的分布式计算技术的兴起，特别是工作站集群系统及其并行编程软件的蓬勃发展和广泛应用，进程迁移由于它在这类系统中所担当的重要角色，会受到越来越多的重视。同时由于网络带宽、传输速度、微处理器等一系列硬件技术的改善和优化，进程迁移在实际应用领域的可行性大大提高，从而使这项研究再次成为热点。进程迁移被分解成两部分的工作：

(1) 在源处理机采集迁移进程的进程状态并转移到目的处理机，并用这些进程状态在目的处理机重建迁移进程，使之从断点处继续运行。

(2) 通知与迁移进程有通信关系的其他进程，重建它们与迁移进程正确的通信连接。

分布计算系统中的进程迁移可以用图 3.6 的概念模型表示，与此模型相关的基本概念是：

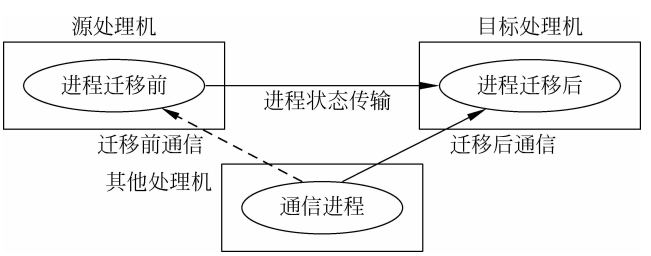


图 3.6　进程迁移的高层概念模型

(1) 进程状态(Process State)。进程是操作系统中的重要概念，一个进程状态由它的用户上下文、寄存器上下文和系统上下文组成，包括：进程数据、代码和堆栈信息，打开文件的状态信息，进程消息，进程执行状态及其自身内核状态信息等。进程迁移时，这些状态信息不一定要全部收集，只需收集支持进程在远程处理机上可以继续运行的完备状态集。在使用检查点技术收集进程状态的系统中，这些状态信息也被称为检查点信息。

(2) 原始主机(Home Node)。在被迁进程的运行操作中，有可能包括一些和主机位置相关的操作，例如数据显示输出，访问本地普通文件等。这时，“原始主机节点”将作为重要概念提出，它是一个进程最初被递交运行的主机节点，有时也称为进程的逻辑运行节点。这些和主机位置相关的操作在递交主机运行和在其他主机上运行时将会得到不同的结果。

(3) 源处理机(Source Node)。源处理机是相对于某次迁移之前，被迁进程所运行的处理节点。由于一个进程在其生命周期内可能被迁移多次，因此，源处理机可能并不是该进程最初递交的原始主机。

(4) 目标处理机(Destination Node)。目标处理机是相对于某次迁移之后，被迁进程所运行的处理机节点。

(5) 远程进程(Remote Process)。在具有递交处理机概念的系统中，运行在非递交处

理机之上的进程被称为远程进程或者是外来进程(Foreign Process)。

(6) 通信关系。在并行程序的运行环境下,各个进程并不是独立运行的,它们可能存在着一定的通信关系。在进程迁移之后,这些通信关系必须能够正确地维护,包括重建通信连接、不丢失消息和保持正确的消息接收顺序。

2. 进程迁移机制

进程迁移机制是如何将一个已运行的进程在源处理机上中断,并将该进程状态迁移到目标处理机上继续运行的过程,即如何实现进程迁移的基本功能。不同的系统可能采用不同的设计细节,最基本的迁移机制将遵照:主机协商→进程冻结→状态收集→状态转移→重新启动等实现步骤。一个典型的进程迁移步骤如下:

1) 迁移协商

由重负载源处理机询问目标处理机是否可以接受迁移进程,或者由轻负载的目标处理机询问重负载的源处理机是否愿意迁移进程。进程迁移开始前,通常存在一个协商过程。在分布式系统中,由源处理机和目标处理机直接协商,主要根据目标处理机的当前状态决定是否适合接收迁移进程;而在集中控制的系统中,通常由中央控制节点根据系统内各处理机的运行状态,决定哪一个处理机适合作为目标处理机接收进程。

2) 创建恢复进程

得到目标处理机的肯定答复后,在目标处理机上创建恢复进程。由于存在进程状态传输,通常需要创建一个恢复进程,该恢复进程负责接收被迁进程的状态。恢复进程最终可能转化为被迁进程新的实例,也可以由该恢复进程再派生一个新进程,并把传输的进程状态加载到该新进程的状态空间中,最终该新进程恢复为被迁进程的新实例。

3) 中断被迁进程运行

一般情况下,源处理机上的被迁进程旧实例都会被中断运行,让进程停滞在一个稳定的状态下,但根据不同的传输优化算法(例如预拷贝方式、惰性拷贝技术等),被迁进程旧实例的中断时机是不尽相同的。

4) 收集源处理机上被迁进程状态

进程状态包括该进程的虚拟存储空间内容、处理机状态(寄存器上下文)、传输状态和一些内核上下文信息等,这些状态信息有一部分存在于操作系统内核中,在没有操作系统内核支持的情况下将不能完整取得。

5) 传输被迁进程状态

即将被迁进程状态传输到目标处理机。如果系统采用了某些状态传输优化措施,可能并不需要将全部进程状态传输到目标处理机,一些进程状态可以在迁移结束后,以惰性拷贝的方式转移到目标处理机。

6) 恢复被迁进程状态

目标处理机上的恢复进程负责恢复被迁进程状态,重建进程实例。目标处理机上的恢复进程负责把传输状态信息恢复到被迁进程的新实例中,将它模拟成被迁进程的原上下文状态。状态恢复可以采用自身恢复或创建恢复的方式。自身恢复是恢复进程用传输的状态信息替代自身的状态信息,恢复进程将最终成为被迁进程的新实例;创建恢复是由恢复进程创建一个新的子进程,并把传输的状态信息加载到子进程的状态空间中,该子进程将最终恢复为被迁进程的新实例,而恢复进程自身在工作完毕后消亡。

7）通告被迁进程的新位置

通知系统内其他进程被迁进程的新位置，并重建迁移中断前的通信连接。被迁进程的通信连接状态也应该属于被迁进程的状态信息，一般系统都会采用特殊的方式单独处理通信连接问题。进程迁移到新的目标处理机后，其定位信息应及时通知外部环境，以便其他进程可以和被迁进程（新实例）重新建立通信联系。迁移进程的通信管理问题在处理并行进程迁移时尤为重要，正确的迁移机制必须保证可以重建通信连接、不丢失消息、维持消息正确接收顺序等。

8）被迁进程恢复运行

当足够的进程状态信息恢复后，被迁进程即可在目标节点恢复运行。根据不同的优化措施，如果源处理机上被迁进程状态均被转移，则被迁进程旧实例可被删除。

9）操作转发

利用转发机制（或利用单一系统视图的性质）保证进程可以在远程处理机执行。当被迁进程在远程处理机上恢复运行后，由于主机状态改变可能导致一系列问题，例如：无法以原来的方式访问文件、设备，或执行某些与位置有关的系统调用等，此时一般需要利用转发机制把这些相关操作转发到原始主机上运行。这就保证了被迁进程远程透明执行。

3.3.2 进程迁移策略：动态负载平衡

进程迁移可以支持动态系统管理和维护、动态负载平衡、系统容错、主人优先使用原则等多项系统功能。实际上，不同的应用需求将导致采用不同的迁移策略。本节主要讨论利用进程迁移支持的动态负载平衡的相关策略，因此，进程迁移的策略转化为动态负载平衡的策略。负载平衡（load balancing）使系统中重负载处理机转移一部分负载到轻负载的处理机上运行，使得整个集群系统中的所有处理机的负载趋向均衡，从而提高系统的整体运行效率。负载平衡系统一般由负载信息管理模块和负载平衡模块构成，它们和迁移机制的关系如图 3.7 所示。

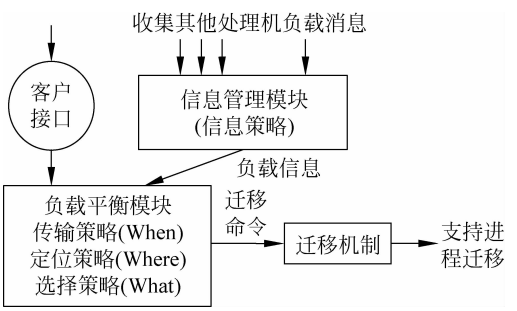


图 3.7 负载平衡策略与迁移机制的关系

1. 信息管理模块

负载信息管理模块主要决定和负载平衡相关的信息策略。为判定何时实施进程迁移，需要把系统内各处理机的负载状况做出标准化的衡量，以此作为负载平衡的依据。对负载信息的衡量及其相关收集机制称为信息策略。信息策略主要包括以下问题。

1）负载信息衡量

衡量一个处理机的负载状况，是执行进程迁移调整负载的基本依据。系统常采用负载指标（load index）对处理机负载进行量化衡量，不同的负载指标定义会得出当前时刻处理机不同的负载程度。对负载指标的研究很多，但没有统一的标准，常用以下的一个或多个负载指标衡量处理机负载状况：CPU 利用率、运行进程个数、背景进程个数、资源利用率、各种资源队列的线性组合、平均队列长度、空闲主存大小等。对于不同的应用系统，负载指标的合理程度不尽相同。

2）信息收集策略

涉及采用什么样的方法收集负载信息。负载信息的收集可以是周期型的，也可以是基于事件触发的。周期型的负载收集机制即每隔一定的时间间隔收集一次负载信息；基于事件触发的方法，是当某类事件（例如进程创建、终止或迁移）发生时，触发一次信息收集。收集信息的方式有集中式或分布式。集中式收集即系统内各处理机获取本地负载信息后，都传送给中央节点；分布式则是在获取本地负载信息后，广播给系统内所有的处理机。集中式收集系统开销较低，但中央节点可能成为系统瓶颈。

2. 负载平衡模块

负载平衡模块决定如何用进程迁移实现负载调整的目的，它依据负载信息管理模块提供的负载信息做出迁移决定。负载平衡策略又可细分为：传输策略，何时进行迁移（When）；选择策略，迁移何种进程（What）；定位策略，将进程迁移到哪一个节点（Where）。

根据信息收集策略，负载平衡也可分为周期型和事件触发型。当负载平衡模块更新负载信息后，发现存在负载不平衡的现象，即可启动各子策略，决定是否需要启动进程迁移进行负载调整。负载平衡模块利用选择策略决定要迁移的对象，利用定位策略决定目标处理机的位置，利用传输策略决定迁移的时机和发起方式。当以上子策略都满足时，即发出迁移命令，利用进程迁移机制实现一次进程迁移操作。负载平衡模块也可以留出用户接口，以支持用户主动提出的负载平衡需求。最常用的负载平衡激发方式有以下几种。

（1）中央服务器触发方式：常用于中央控制方式的负载平衡系统，当系统的中央控制节点发现存在系统不平衡状态时，触发进程迁移。

（2）发送者触发方式：当某处理机发现自身过载时，寻找负载轻的目标处理机，并触发进程迁移。

（3）服务者触发方式：当某处理机处于轻负载状况，主动要求接收其他重负载节点的进程，分担负载。

（4）对称触发方式：是发送者激发方式和服务者激发方式的综合，要求发送者和服务者均参与负载分配的活动，在系统负载轻时，发送者激发方式有效，反之，服务者激发方式有效。

（5）自适应触发方式：是一种优化的方法，它根据系统状态的变化改变某些参数甚至策略来适应系统负载的变化。

3.3.3　进程迁移的实现

1. 进程状态收集和恢复

进程状态收集和恢复是两个独立而又关联的过程。系统将通过网络将收集的进程状态传输到目标处理机上进行恢复，恢复后的进程将从中断的断点处继续运行。完整和正确的状态收集是状态恢复的前提。

1）进程状态收集

进程迁移最重要、最基础的过程是进程状态的成功收集和恢复。从操作系统的观点来看，在传统的 UNIX 系统中，进程被抽象为上下文信息（Context），这些进程上下文信息可以看成进程的状态信息集。一个进程的上下文是由用户空间的内容、硬件寄存器的内容以及

与该进程有关的数据结构组成。更严格地说，进程上下文由它的用户级上下文、寄存器上下文和系统上下文组成。

保存进程状态以供进程迁移后恢复用，并不是所有这些信息都需要保存，实际上，只需要保存用户级上下文、寄存器上下文和部分系统上下文。因为系统级上下文通常是操作系统在创建和管理进程时所分配、使用和维护，它们只是操作系统调度进程和管理资源所必需的框架，其中的相当一部分对于进程的运行来说是透明的，所以没有记录和恢复的必要。

目前的状态收集的接口形式可以归结为三种类型：内部状态收集、外部状态收集和触发式状态收集。

(1) 内部状态收集。

进程主动调用状态收集函数对自身状态进行收集。这种方式将状态收集函数直接写入程序代码中。它的主要缺点是迁移过程由用户参与，降低了进程迁移的灵活性和系统可控性。内部状态收集的另一个难点是在状态收集过程，用户进程始终处于运行状态，其状态信息也随之处于不断变化的情况，因此必须采取特殊的手段保证收集状态的一致性。

(2) 外部状态收集。

这种方法将需要迁移的进程挂起，然后通过操作系统提供的特殊系统接口对该进程进行状态收集。利用这种方法可以实现对自身或其他进程的状态收集，它无须用户介入，不需对原有进程的修改，甚至可以对只有二进制文件格式的进程执行状态收集，因此其透明性、灵活性、可控性都好于第一种方法。但是该方法只能在操作系统的核心级实现，而且由于对进程的状态收集不是由进程自身而是由辅助进程完成的，所以也一定程度增加了系统的负担。

(3) 触发式状态收集。

由信号触发导致进程对自身进行状态收集。这种方式中，需要在程序中设置特殊的迁移信号及信号处理函数，当迁移进程接收到迁移信号时，就会进入相应的信号处理函数进行自身状态收集。这种方法一般不需要对应用进程进行直接修改，但需要重新编译和连接进程迁移函数库，将信号和信号处理函数连接到应用进程。该方法的实现可以采取操作系统内核实现，也可以采用用户级实现的方法。但是由于在状态收集过程中进程仍然处于运行状态，和内部状态收集的方法一样，该方法也需要进程状态一致性的处理。

为减少状态收集的信息，系统将选择进程包含较少状态信息的时刻执行状态收集，这个时刻应该是进程在用户态运行的时刻。

2) 进程状态传输

进程状态成功收集，并在目标节点上恢复的过程之间存在一个状态传输过程，即将这些收集的状态通过网络传送到目标节点上以备恢复。在不支持分布式共享存储器或远程内存访问的通用操作系统中，状态传输一般通过网络进行传输，最常用的方法是先将收集的进程状态写入检查点文件，再将检查点文件通过网络传送到目标主机，然后恢复进程在目标处理机上从检查点文件中读出进程状态进行恢复。这种方式可对检查点文件进行随机访问。

3) 进程状态恢复

状态恢复是将收集的进程状态信息在新目标节点恢复的过程，它基于状态收集的正确性保证。如果收集的状态信息足够完整恢复进程上下文状态，则进程可以在新处理机上重

建，而重建进程实体正是中断前的运行状态，因此该进程将可以从断点处继续运行。状态恢复的方式可以分为自身恢复和创建恢复两种。自身恢复是恢复进程利用收集的状态信息将自身状态进行替换，这种恢复方式的优点是效率较高，但是整个过程缺乏灵活性和可控性，对异常和出错处理能力较弱。创建恢复即由系统调用恢复进程创建迁移子进程并将状态检查信息加载到子进程的框架中，将它模拟成原应用进程的状态。这种恢复方式具有较高的灵活性和可控性，对异常和错误的处理能力也强，但付出的代价是系统耗费相对增大，效率有所降低。

2. 转发机制

转发机制是针对进程行为中与位置相关的操作，与位置相关状态主要包括一些与位置有关的 I/O 资源状态和一些执行内核行为的内核堆栈信息，如文件管理结构和文件句柄。这些资源无法迁移到目标处理机。进程迁移到目标处理机后，对这些资源的操作只能转发到原始处理机上执行。而寄存器内容、虚存管理结构、虚存区表等都是与位置无关的资源，将随进程一起迁移到目标处理机上恢复执行。

对进程的资源做了区分后，还必须对进程的行为进行区分，对于使用系统调用方式访问内核的类 UNIX 系统来说，就是要区别对待与位置相关的系统调用及与位置无关的系统调用。当迁移后的进程在远程处理机上执行某一与位置相关的系统调用时，系统将截取该操作，通过网络转发到原始处理机上。这时原始处理机上应运行一个和迁移进程对应的伺服进程，伺服进程负责接收转发的操作，在本地执行完毕后，再将运行结果传送回目标处理机。

图 3.8 表示 VDPC 的转发机制。以访问本地文件操作的进程迁移为例，说明转发机制如何工作。实际上，分布式文件系统也可以随进程迁移到目标处理机上执行。当进程 A 迁移时，与位置无关的进程状态和该进程一起迁移到目标处理机，系统同时在迁移进程的原始处理机派生一个伺服进程(A-伺服)，并将迁移进程遗留在原始处理机上的与位置相关的进程状态(这里是迁移进程打开文件的状态信息)恢复到该伺服进程中。A-迁移进程在目标处理机上恢复后，当执行本地文件操作时，系统截获该系统调用，将其传送给 A-伺服进程，并等待接收处理的结果。因为 A-伺服进程保存有该迁移进程打开文件的状态包括文件读、写位置，因此该机制可以有效地保证进程远程执行的文件操作和在本地执行的结果一致。

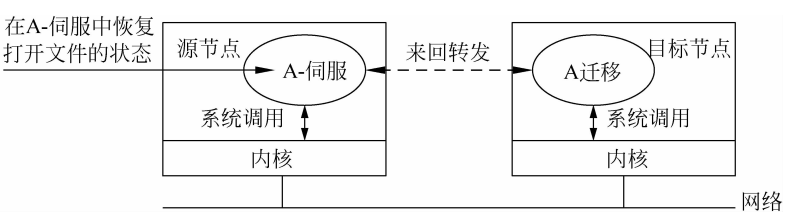


图 3.8　VDPC 转发机制

3. 通信恢复

由于进程迁移后位置改变，可能导致通信丢失问题。进程迁移后通信丢失问题概括为以下三类用户消息。

1) 被迁进程到的新地址

用户进程 P_1 被迁到新的节点处理机后，采用了新的地址，将会导致发送给被迁进程和

发自被迁进程的消息丢失，如图 3.9 中消息 M_1 和 M_2。M_1 是由非迁移协同进程 P_2 以旧地址（地址 1）发送给被迁进程 P_1 的消息，由于以地址 1 为通信地址的进程已不再存在，M_1 消息丢失。进程迁移后，获得新地址 2，以地址 2 发送给非迁移协同进程 P_2 的 M_2，由于接收方等待的地址仍然是旧地址 1，消息 M_2 不被 P_2 接收，也造成消息 M_2 丢失。解决这个问题的办法有两个。

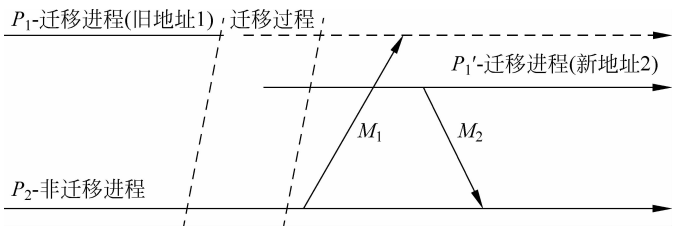


图 3.9　进程通信堆栈改变导致的问题

（1）建立进程地址映射表。在用户进程发送/接收消息时，先进行地址匹配，引导以旧地址为目的地的消息转向新的地址。

（2）采用特殊路由方式。让进程迁移后，通信地址不变，而在路由表中增加从旧地址到新地址的映射。

2）保证不丢失任何消息

另一类消息是在迁移前发出但尚未被目标进程接收到的消息，以及在迁移过程中发送的消息，这类消息称为“中途消息”，如图 3.10 所示的 M_3 和 M_5。现在的问题是要保证“中途消息”不被遗失。保证“中途消息”不丢失的常用方法有两种。

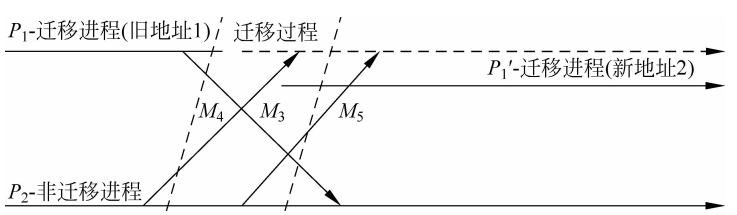


图 3.10　中途消息

（1）消息驱赶方法。将所有的中途消息驱赶到目标进程之后，再进行进程迁移过程，并保证在迁移过程中不再发送任何“中途消息”。

（2）消息转发方法。进程迁移后，被迁进程旧实例不中断执行，中途消息仍然发到旧地址，由被迁进程旧实例将消息转发到被迁进程新位置。被迁进程旧实例可以一直运行担任转发工作，也可以在保证不会再有以旧地址为目的地址的用户消息后，终止运行。在实际实现中，以上两类方法还需要一些辅助的消息处理机制，才能较为严格地实现中途消息不被丢失。

3）维护消息正确顺序

在采用消息转发方法处理“中途消息”的系统中，由于两个进程之间的消息可能通过不同的通信信道到达，因此在使用不可靠传输协议的系统中，应通过一些特殊的手段保证消息的接收顺序和发送顺序的一致性。维护消息正确顺序有两层意思：第一层是长消息分片，

要确保分片顺序；第二层是维护不同消息的先后顺序。维护消息片顺序的方法如下。

① 增加标志信息。在消息头中添加必要的标志信息。例如：在消息头中添加消息总长度信息，在消息片描述符中添加消息 id 标识以及消息片次序 sid。这种方法执行效率较高，但需要修改并行编程环境源码且给正常的消息通信带来较大的附加开销。

② 原子通信。保证消息通信的原子性，即保证同一消息的不同消息片使用相同的通信信道传递。该方法在基于消息驱赶的系统中可以得到较好的保证，信道消息驱赶机制可以把该通信信道内的消息全部驱赶到目标进程，不会产生同一消息的不同消息碎片使用不同的通信信道传递的现象。这种方法实现简单，但不够灵活且效率较低。

维护不同消息的先后顺序的方法如下。

① 消息附加消息序号。进程间的通信消息附加消息序号以实现排序。

② 采用特定机制保证处于迁移临界区消息的正确顺序。即保证迁移开始发送且达到目标进程的消息→迁移过程前或过程中发出但目标进程视为接收的消息（中途消息）→迁移过程完全结束后发送的消息顺序处理。

4. 进程迁移算法分类

根据在迁移过程中其他非迁移进程的参与方式，可以分为以下三类。

1）异步迁移算法

异步迁移算法允许非迁移进程在迁移过程中继续运行，只有被迁进程被中断进行相关的操作（但是如果进程间的通信是通过直接方式连接的，与被迁进程正在通信的非迁移协同进程也要被中断，暂停通信）。异步迁移算法需要通过直接修改并行编程环境源码的方式实现。使用异步迁移算法可以得到较好的执行效率，但是它的突出缺点是：由于对原有并行编程环境做了较大的修改，和原有环境的兼容性不好，如果底层并行编程环境版本升级，必须对现有支撑环境重新编码，不能方便地移植。

DynamicPVM 是采用异步迁移算法的典型系统，它利用进程迁移实现动态负载平衡与调度。对于使用间接通信（经路由器）和直接通信，DynamicPVM 使用了两种不同的处理方式。

（1）间接通信的迁移协议。

DynamicPVM 在进程迁移前后使用相同的标识符 Tid，系统查询 PVM Daemon 中的路由表，定位进程的位置。进程迁移前，系统将广播迁移进程的路由信息。修改了路由表信息后，相关消息将发送到被迁进程的新位置（在迁移进程尚未在新位置恢复运行之前，这些信息由新位置的 Pvmd 缓存）。在之前的中途消息将采用消息转发和排序机制转移到被迁进程的新位置。可以看出，所有使用间接方式和被迁进程通信的非迁移协同进程，不需要中断来同步等待迁移过程结束。

（2）直接通信的迁移协议。

由于进程迁移过程中，所有与被迁进程采用直接通信方式的非迁移协同进程，都将被迫中断。DynamicPVM 对直接方式通信采用了不同的协议，即用消息驱赶机制清空相关信道的中途消息，待进程迁移到目标节点后，再重建所有的直接连接。

DynamicPVM 完全修改了 PVM 应用消息的格式，将消息长度、消息标识和消息片标识

写入消息头部，实现消息顺序重整。这种消息转发和排序机制，加大了通信开销。

2）同步迁移算法

同步迁移算法在迁移过程中所有进程（包括非迁移协同进程）都被挂起，进程之间需要同步来清空通信信道中的中途消息，所有进程均要阻塞等待到迁移事件完成后，才能从中断处继续运行。同步迁移算法由于算法简单，其并行进程迁移算法和相关消息处理都可以在并行编程环境的外层实现，这意味着这些并行进程迁移算法是独立于底层并行编程环境的版本特征的，具有较好的可移植性和易于实现。但需要中央控制管理进程参与，所有进程都被迫中断，等待迁移过程的结束。

CoCheck 系统采用标准的同步迁移算法，其过程分为中断、同步、收集和恢复 4 个阶段。算法需要通过中央（集中）控制管理进程（RM）协助，迁移开始由 RM 发出同步信号给所有进程，所有进程中断执行任务，进入同步等待。所有中途消息都被驱赶到接收进程，并作为接收进程的一部分状态信息，清空信道。在状态收集和恢复阶段，所有进程收集的状态以磁盘文件形式保存。进程恢复运行后，重新加入并行运行环境，注册成新的 PVM 任务并获得新的标识 Tid。这时还需 RM 广播地址映射表，将被迁进程的新地址通知给所有与之有关的非迁移协同进程，使消息能发送到正确的位置。

3）类异步迁移算法

类异步迁移算法中尽管非迁移协同进程也像在同步算法中那样被中断，但是它允许非迁移协同进程在迁移过程中继续运行，只是在迁移过程的某些时刻进行简单的协调工作。这类算法是以上两类算法有效结合而派生出的一类新的迁移算法，该类算法在并行编程环境上层实现迁移，修改和优化传统同步迁移算法，使其接近异步迁移算法的效率，却避免了异步算法需修改并行编程环境源码、实现复杂等缺点。

VDPC（Virtual Dynamic Personal Cluster）集群系统是北京航空航天大学分布与移动计算实验室的一个研究项目，使用的是类异步迁移算法（MFQA）。无须中央（集中）管理进程，易于管理与控制。非迁移进程只需短暂中断一次，不必同步等待，大大减少了协调开销。借用包裹技术将进程迁移需要的附加信息分布于各个应用进程，独立维护。源节点上的被迁进程存活到进程恢复之后，负责中途消息的转发。MFQA 提供了 4 种机制，保证进程迁移过程中消息不丢失和正确的传送顺序。

（1）地址映射表。

每个应用进程独立维护一个进程新旧地址映射表。进程通信时先查询此映射表，获得通信进程的当前位置，才进行通信。解决迁移进程新位置的识别问题。

（2）消息驱赶机制。

消息驱赶机制利用了通信的 FIFO 特点。每个进程向其通信通道发送一个特定消息后不再向该信道发送任何消息。当接收方收到此特定消息后，表明此前在该信道上发送的消息已全部到达接收方。

（3）消息转发机制。

将迁移开始之后才到达被迁进程的旧地址的消息转发到被迁进程的新地址。这个转发工作是由暂活在源节点的被迁进程旧实例完成的。这个转发机制只在迁移临界时段短暂存在。地址映射表、消息驱赶机制和转发机制保证不会丢失任何消息。

(4) 缓存优先匹配机制。

两进程之间的消息可能从不同的信道到达,此时无法保证和利用其 FIFO 特征。转发来的消息应早于其他应用消息,缓存优先匹配机制将转发来的消息缓存在内存中。被迁进程恢复正常运行接收消息时,先匹配缓存中的消息,从而保证了迁移进程中消息的正确传送顺序。

3.4　分布式对象

基于对象的技术在开发非分布式应用程序方面已展示了它的价值。对象重要的特征之一是它的封装性,通过定义良好的接口对外界隐藏其内部结构。这种机制保证:只要保持对象接口不变,可以方便地替换和修改对象。面向对象的编程范型是从结构化程序设计发展过来的,它能更好地刻画世界现实。面向对象编程范型的特点是抽象、封装、继承和多态性。这一节简要介绍分布式对象的基本概念和特点。

3.4.1　对象生成与适配

在分布式系统中对象可能以多种形式出现。一个对象由两部分组成:状态和行为。状态是对对象属性的描述,通常用各种数据结构表示。外界对对象的属性可以获取(Get)和设置(Set),但单向属性只能获取。行为是访问对象状态的方法,是一个过程。方法通过接口向外界提供,外界对对象的服务请求只能经过接口实行。

1. 对象生成

最常见的一种对象形式是直接与语言级对象相关联,称为面向对象的编程范型。由面向对象编程语言调用的对象称为编译时对象。编译时对象是对象系统的源泉。当前常用的面向对象的编程语言有 C++、Java、VB 等。在这些语言中支持对象的结构是类(Class)。类定义了对象的结构,规定了对象的状态和方法,对象是类的实例。下面是一个堆栈类,它定义了一个整数堆栈对象:

```
Class Stack{
    int * number;                          //堆栈的项数, * number 为堆栈指针
    int array[];                           //堆栈内容
    Stack (int limit){
        array = new int[limit];            //创建一个数组
        * number = 0;                      //堆栈为空
    }
    void push(int x){
        array[ * number++];                //将整数 x 压入堆栈
    }
    int pop(){
        return array[ - - * number];       //递减 number 后,取出堆栈项
    }
    Boolean isEmpty(){
        return * number == 0;              //如果堆栈为空,布尔值为真
    }
}
```

堆栈是由一个数组实现，数组的长度，即堆栈的深度是 limit，堆栈项的指针是 number。对该堆栈规定了三个过程（方法）：压入 push、弹出 pop 和检查堆栈是否为空。

对象可以是持久的。持久对象无论当前是否位于服务器进程的地址空间内，都始终存在。这意味着当前管理持久对象的服务器在退出运行之前，先把持久对象存储到固定存储器；之后，重新启动的服务器可以从固定存储器读到自己的地址空间，对外部的访问进行处理。

暂时对象只存在于管理该对象的服务器运行期间，服务器退出运行后，对象就不复存在。

2. 对象适配器

对象适配器是一种对已有的对象实现重用机制，包括用各种编程语言实现的对象的重用。它也能将函数库进行包装，使得函数库对外像一个对象，外部就像引用对象的方法一样调用这些函数。对象适配器并不产生对象，而是对现有对象进行接口的转换。例如，用 Java 的枚举类 Enumeration 设计一个简单的 MessageApplication 对象，显示对象接收的元素。MessageApplication 对象如下：

```
import java.util. * ;
public class MessageApplication{
   public void showAllMessage(Enumeration enum){ //定义枚举数据结构
      Object msg;
      While enum.hasMoreElemets(){                //确定枚举数据结构是否还有元素的方法
         msg = enum.nextElements();               //读取枚举数据结构的下一个元素
         System.out.println(msg);
      }
   }
}
```

MessageApplication 对象提供两个方法：确定枚举数据结构是否还有元素的方法 enum.hasMoreElemets()和读取枚举数据结构的下一个元素 msg = enum.nextElements()。客户端程序可以直接引用对象 MessageApplication。后来，Java 引入了迭代器类 Iterator，可以定义一个对象，替换 Enumeration。但另一种处理办法是保留 Enumeration，使用 Iterator 的客户程序通过对象适配器可以访问 Enumeration，Iterator 对象适配器 IteratorAdaper.java 如下：

```
import java.util. * ;
public class IteratorAdapter implements Enumeration{
   private Iterator iterator;
   IteratorAdapter(Iterator iterator){              //转换接口
     This.iterator = iterator;
   }
   public Boolean hasMoreElements(){
      return iterator.hasNext();
   }
   public ObjectnextElement()
       throws NoSuchElementException{
         return iterator.next();
       }
}
```

通过对象适配器引用已有的对象，是独立于编写分布式应用程序所用的编程语言，特别是可以使用由各种语言编写的对象来构造应用程序，很多基于对象的分布式计算系统都使用对象适配器。

一个对象适配器可以控制一个或多个对象。由于一个对象服务器能支持要求不同激活策略的对象，因此在一个对象服务器中同时驻有多个对象适配器。图 3.11 表示对象服务器中驻有两个对象适配器，一个控制一个对象，另一个控制两个对象。

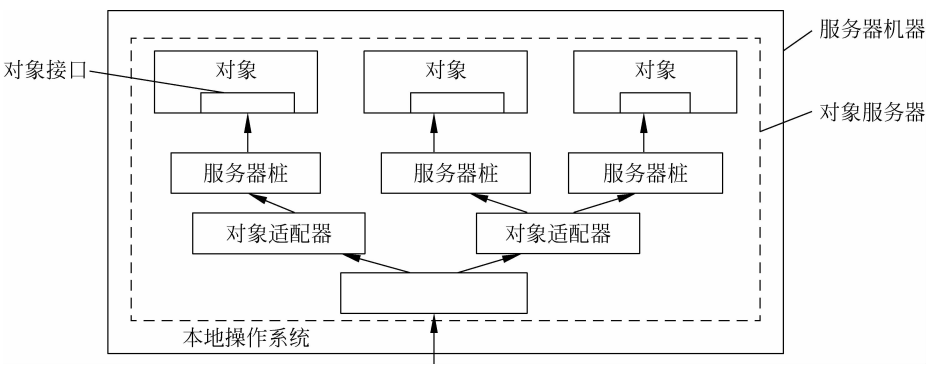


图 3.11　驻有两个适配器的对象服务器

3.4.2　分布式对象的特点

传统面向对象程序设计环境 OOP(如 C++、Smalltalk)，其对象与访问该对象的程序只能存在于同一进程中，只有相关程序设计语言的编译器才能创建这些对象并感知它们的存在，外部进程无法了解和访问这些对象。这意味着在分布式客户/服务器应用中，客户进程不可能直接访问异地服务器中的常规对象。为了解决这个问题，人们提出了分布式对象的概念。分布式对象是一些独立的代码封装体，它向外提供一个包括一组属性和方法的接口，远程客户程序通过方法调用来访问它。分布式对象具有如下特征。

① 分布式对象位于网络何处、使用何种编程语言、编译器如何创建分布式对象以及它们运行于何种硬件和操作系统平台之上，对客户来说都是透明的。

② 每个分布式对象都定义有清晰的访问接口，分布式对象之间只能通过这些预先定义的接口进行访问，这些接口构成连接客户程序和服务器程序的协议。

③ 面向对象的多层客户/服务器计算模型组织各种分布式对象。任何分布式对象都可向其他对象提供服务，也可向它们请求服务，客户与服务器的角色划分是相对的或多层次的。

④ 分布式对象具有动态性，它们可以在网络上到处移动。

1. 对象与客户绑定

一个进程除了通过对象的接口来调用它的方法外，没有其他途径能访问或操纵对象的状态。将接口与实现这些接口的对象分离开来对于分布式对象系统是非常重要的。由于这种分离，接口可以放在一台机器(一般是客户机)上，对象本身驻在另一台机器(服务器机器)上。图 3.12 是客户与远程对象的一般组织。当一个客户绑定到一个分布式对象时，该对象接口的一种实现被加载到客户进程的地址空间，这种实现称为客户桩(Stub)，或称代理

(Proxy)。例如,编写客户应用程序时调用对象接口。客户进程通过客户桩接口传给客户桩,客户桩将客户进程的方法调用编码成消息送给服务器桩(或称骨架 Skeleton);解码服务器应答信息,将调用结果返回给客户进程。服务器桩解码客户的消息,实现对对象方法的调用;将调用的结果编码成消息送往客户端。

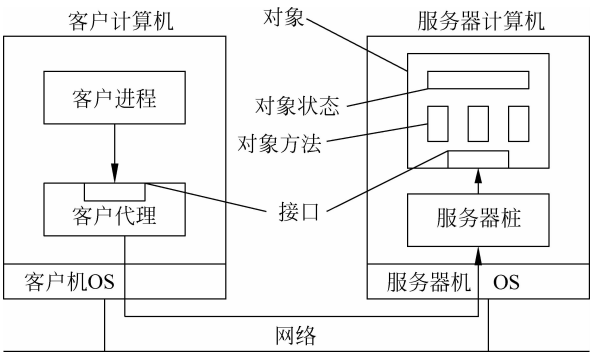


图 3.12 客户与远程对象的一般组织

2. 远程方法调用

使用预先确定的接口调用的方法称为静态调用。静态调用要求在编写客户语言程序时就已经知道对象的接口。这就意味着,若接口发生变化,客户应用程序必须重新编译,才能使用新的接口。现实情况需要一种更为动态的方式进行方法引用。这是在客户应用程序运行过程中建立方法调用,这种做法称为动态调用。

基于对象的分布式计算系统的体系结构和方法引用将在第 11 章进行更详细的讨论。

本章小结

本章讨论了分布式进程、进程远程执行、进程迁移、分布式对象和软件移动代理等计算范型。

3.1 节从支持分布式计算出发,讨论了分布式进程的结构和支持客户/服务器计算模型的多线程进程结构。

3.2 节讨论了客户应用进程的远程执行。对资源贫乏的手持计算设备,需要将客户的应用发射到网络上空闲或轻负载节点上,依赖网络资源完成其计算;对于复杂的计算,如海洋模型的 e-科学问题,客户(科学家)需要将应用发送到集群或计算网格的众多节点上执行。

为了达到分布式系统的动态负载平衡或其他目的,进程需要在分布式系统节点间移动,3.3 节讨论了进程迁移问题,包括进程迁移的模型、过程、转发机制和通信一致性和连贯性。

3.4 节简要介绍了分布式对象,第 11 章还会对分布式对象的组织结构进行详细讨论。

习题

3.1 什么是重量进程、中量进程和轻量进程?比较它们的优缺点。

3.2 假定需要的数据存放在服务器的主存中,服务器要花费 15ms 来接收、调度请求

和其他必要的处理工作。如果数据在服务器的磁盘中，要求 75ms 读磁盘，在这种情况下单线程服务器完成一次数据请求需要多少时间？

3.3　如果客户访问服务器读取三个对象，两个缓存在服务器主存中一个在服务器的硬盘中，访问主存和磁盘的时间如题 3.2，多线程服务器完成三个对象读取需要多少时间？

3.4　创建分布式进程要考虑哪些问题？目标机选择策略是什么？

3.5　分布式进程执行环境如何建立？

3.6　进程远程执行的含义是什么？它用在什么场合？

3.7　什么是远程执行逻辑机模型？对逻辑机模型的要求是什么？

3.8　进程迁移为什么没有被当前大多数操作系统所支持？

3.9　进程迁移可以用在什么场合？

3.10　进程迁移为什么要有转发机制？

3.11　考虑某个进程 P，它请求访问与自己位于同一台机器上的本地文件 F。进程 P 迁移到另一台机器上后，它还需要访问文件 F。如果文件 F 与机器是紧密绑定，进程迁移后如何实现对文件 F 调用？

3.12　进程迁移如何保证与其他进程的通信不被丢失？

3.13　何为异步进程迁移算法？何为同步进程迁移算法，它们的优缺点是什么？

3.14　VDPC 的类异步进程迁移算法采用什么措施保证进程迁移过程中消息不丢失和正确的传送顺序。

3.15　比较进程远程执行与进程迁移两种机制。

3.16　对象适配器的作用是什么？

3.17　叙述分布式对象的特点。

3.18　客户桩和服务器骨架的实质是什么？它们在远程方法调用中起什么作用？